

Redynis: Traffic-aware dynamic repartitioning for a distributed key-value store

Vineet John

David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
vineet.john@uwaterloo.ca

ABSTRACT

Most modern data stores tend to be distributed, to enable the scaling of the data across multiple instances of commodity hardware. Although this ensures a near unlimited potential for storage, the data itself is not always ideally partitioned, and the cost of a network round-trip may cause a degradation of end-user experience with respect to response latency. The problem being solved is bringing the data objects closer to the frequent sources of requests using a dynamic repartitioning algorithm. This is important if the objective is to mitigate the overhead of network latency, and especially so if the partitions are widely geo-distributed. The intention is to bring these features to an existing distributed key-value store product, Redis[2].

Keywords

dynamic repartitioning, key-value store, data placement

1. INTRODUCTION

The objective of this project (Redynis) is to design a shared-something distributed architecture atop an existing key-value store that dynamically repartitions tuples based on traffic metrics.

Redynis takes its roots in intelligent data placement, and in essence, attempts to delegate the data ownership to the distributed key-value store instance that is closest to the most frequent sources of a request, by implementing a web-service as an intermediary layer to the key-value store on each node.

2. PROBLEM STATEMENT

The aim is to solve the problem of having to make frequency remote requests for local node cache misses. This needs to be solved in a manner that allows for a more usage-heuristic based dynamic repartitioning of the tuples, and build a framework that intelligently repartitions tuples for a distributed key-value store.

The motivation of Redynis is three-fold:

- Reduce the network latency by dynamic repartitioning of the key-value tuples based on usage-traffic heuristics i.e. maximize the number of hits on the local data store.

- Leveraging the same usage-traffic heuristics to selectively purge stale data.
- Optimizations need to be non-blocking so as to not interfere with the regular execution of fetch requests.

Redynis is built with the purpose of implementing these features to reduce cross-node request latency.

3. SIMILAR WORK

This section lists the previous work done to solve the same or a similar problem to the one described in the previous section.

Attempts to identify ideal methods to dynamically partition data already exist. Two of them, SWORD[9] and AdaptCache[4], rely on hyper-graphs to model the database workloads, and base the repartitioning decisions off this model. SCHISM[5] relies on graph partitioning to find a predicate-based explanation for the ideal partitioning strategy. E-store[10] relies of a strategy of skewed placement, where the data placement decision is taken based on usage heuristics, while avoiding replication.

Redynis, however, is implemented in a manner that avoids expensive graph traversals and is able to log usage heuristics and perform a usage analysis for a key in constant time. In addition, since Redis is widely used in a lot of industrial tech stacks[3], it makes for an easier deployment strategy, compared to switching over to a new system. Also included, is a web API for ease of data access, which minimizes the development overhead of having to implement a language specific client to interact with the key-value store. Any other key-value store can be also swapped in, in place of Redis, without any changes to the client's view of the architecture.

4. MODEL ASSUMPTIONS

The following assumptions are made, with respect to the system architecture and the problem statement:

- The load balancing layer on the application servers hosting the web-service ensures that the request from clients is served by the application server closest to the client. This problem has already been solved by host resolution techniques by a DNS setup[8].
- The nature of the workload, as is the case with most key-value stores, is predominantly read requests.

- The network of nodes is geo-distributed
- Minimal memory usage on each of the nodes is a desirable property
- $size(value) \gg size(key)$

5. SYSTEM ARCHITECTURE

This section elaborates on the components the system is comprised of, and explains the points of interaction between them. **Figure 1** shows a graphical representation of the architecture.

5.1 Components

The components of the architecture are listed below:

- **Web service layer:** This layer of abstraction over the key-value store, is deployed on the application server nodes. It receives requests, reads placement details from the metadata layer and acts accordingly.
- **Data Layer:** This is the underlying in-memory data-store which the objects are primarily stored in. There is a single key-value store instance running on each of the nodes.
- **Metadata Layer:** This is a smaller key-value store for metadata, which is a separate cluster running on the same nodes as the actual key-value stores. It stores the key metadata, like current placement, usage heuristics and recency of access.
- **Placement Daemon:** This continuous process keeps running offline in periodic intervals to repartition the keys, based on the placement strategy described in **Algorithm 3**.

5.2 Component Interaction

Component Interaction can be enumerated as below:

- The web service deployment instances are agnostic of each other.
- The metadata layer and the caching layer are agnostic of each other.
- A given web-service on node can initiate a key-value store request call on any of the instances in the cluster of nodes.
- The data placement daemon is agnostic of the web-server layer. It merely reads from the metadata layer and enforces changes to the key-value store instances.

6. KEY CONCEPTS

This section contains a description of the key concepts that the dynamic repartitioning strategy, is heavily dependant on, including the ‘Ownership coefficient’ and the data format in which metadata for each key is maintained.

6.1 Ownership coefficient

The ownership coefficient (H) determines which nodes need to have a local copy of a particular key/object.

During the analysis phase of the data placement daemon, the key usage for each node is calculated using equation 1:

$$g(O, x) = \text{count}(\text{accesses on object } O) \text{ by node } x$$

$$f(O, x) = \frac{g(O, x)}{g(O, \forall \text{nodes})} \quad (1)$$

If equation 2 holds true, then node ‘x’ is deemed eligible to possess a replicated copy of object O.

$$f(O, x) - H \geq 0 \quad (2)$$

The above conditions operate under the constraint defined in equation 3

$$H - \frac{1}{n} \leq 0 \quad (3)$$

where ‘n’ is the number of nodes in the architecture. This constraint is defined to avoid host starvation of key ownership, especially for cases in which hosts might have close to equivalent access metrics, and result in undesired deletion of keys from nodes.

6.2 Metadata format

The data object used to store the metadata for each tuple is given below.

```
{
  'totalAccessCount': 17,
  'hosts': [
    'node-1',
    'node-3'
  ],
  'hostAccesses': {
    'node-1': 9,
    'node-2': 3,
    'node-3': 5
  },
  'lastAccessedDate': 1480725771235
}
```

hosts is a hashed set, *hostAccesses* is a data-dictionary, and the numeric values are positive integers. *lastAccessedDate* denotes when the key in question was last accessed, in terms of milliseconds elapsed since the epoch.

7. ALGORITHMIC APPROACH

This section describes the algorithms being used to implement the architecture, including fetching, storing and repartitioning tuples.

7.1 Fetching tuples

Described in **Algorithm 1**.

7.2 Storing tuples

Described in **Algorithm 2**.

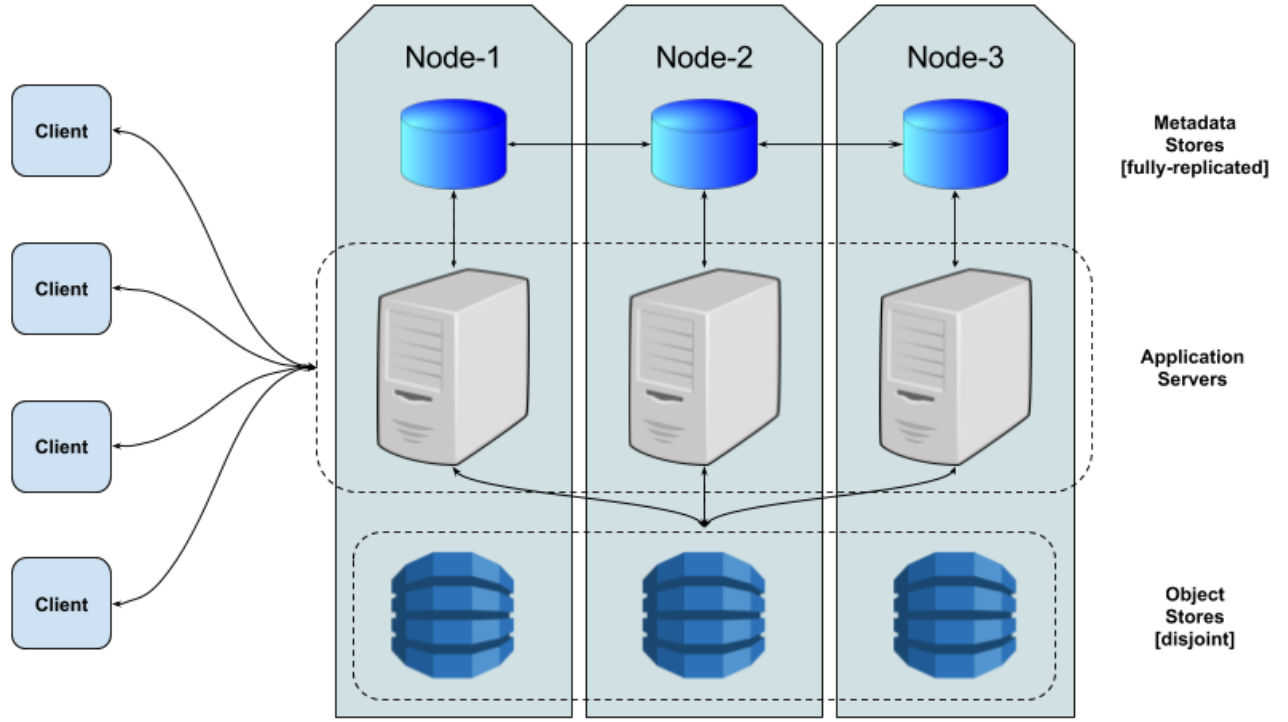


Figure 1: System Architecture

Data: key
Result: value/null

```

query metadata data for key
if metadata == null then
  | return null;
else
  | owner_hosts = metadata.hosts
  | if current_host ∈ owner_hosts then
  |   | make local request to get data
  | else
  |   | make remote request
  |   | (incurring additional latency)
  | spawn async thread and collect access metrics
  | return value to user

```

Algorithm 1: Fetching values

Data: key, value
Result: success = true/false

```

query metadata data for key;
if metadata == null then
  | store new key and value locally
  | generate metadata object
  | post metadata object to metadata layer
else
  | owner_hosts = metadata.hosts
  | if current_host ∈ owner_hosts ∧
  |   length(owner_hosts) == 1 then
  |   | key is only present at current-host
  |   | post value locally
  | else if current_host == write-serializer then
  |   | post value to owner-hosts
  | else
  |   | relay store request to write-serializer node
  | end

if no-exception-thrown then
  | return true
else
  | return false

```

Algorithm 2: Storing values

7.3 Repartitioning tuples

Described in **Algorithm 3**.

Data: master-metadata-host

Result: null

initialize $H = \text{ownership.coefficient}$

initialize *owner_hosts*

initialize *delete_hosts*

```
for  $key \in \text{all\_keys}$  do
     $\text{key\_accesses} = \text{key.metadata.total\_accesses}$ 
     $\text{current\_hosts} = \text{key.metadata.hosts}$ 
     $\text{owner\_accessmap} = \text{key.metadata.accessmap}$ 
    if  $\text{now} > (\text{key.metadata.hosts} - \text{expirytime})$  then
        delete  $key$  from  $\text{current\_hosts}$ 
        delete  $key$  from  $\text{metadata}$ 

    for  $\text{hostaccess\_pair} \in \text{owner\_accessmap}$  do
         $f = \frac{\text{hostaccess\_pair.accesses}}{\text{key\_accesses}}$ 
        if  $f \geq H$  then
            add  $\text{hostaccess\_pair.host}$  to  $\text{owner\_hosts}$ 
        else
            add  $\text{hostaccess\_pair.host}$  to  $\text{delete\_hosts}$ 
    end

     $\text{new\_hosts} = \text{owner\_hosts} - \text{current\_hosts}$ 
     $\text{obsolete\_hosts} = \text{current\_hosts} \cap \text{delete\_hosts}$ 

    add new hosts and delete obsolete hosts from
    metadata
end
```

Algorithm 3: Placement Algorithm

8. TEST SETUP

This section describes the details of the test setup, including the test bed and the nature of the experiments conducted.

8.1 Testbed

8.1.1 Service Infrastructure

The testing for this experiment was done on a cluster of 3 nodes, with 12 CPU cores and 16 GB of main memory. Each of these nodes contains a deployment setup as follows:

- RedynisService [7]
- Redis instance (as the actual key-value store)
- Redis instance (as the metadata store)

Of these, one of these nodes is configured to be the master propagator, in order to serialize write transactions and ensure correctness of value data across the Redis instances.

8.1.2 Placement Infrastructure

A single node will be running a continuous execution for RedynisDaemon [6] The node's hardware specifications the same as the nodes running the service infrastructure (8.1.1).

8.2 Workloads

YCSB workloads for RESTful web-services was used to benchmark this experimental setup.

The tests run were permutations of the below configurations:

- Workload Read requests (%) ranging from 100(all reads) to 50(write-heavy)
- Uniform key-value access distribution vs Skewed key-value access distribution

The uniform distribution workload accesses all the tuples an equal amount of times, whereas the skewed distribution workload accesses is a zipfian distribution that requests 10% of the data items 90% of the time.

All of the workloads have been run on a uniform set of 100,000 total requests. To simulate a widely geo-distributed network of nodes, the incurred latency for making a request to a remote node is simulated to be 100ms[1], whereas there is no incurred penalty for making a local request.

9. EXPERIMENTAL RESULTS

The experimental results for uniform distribution of object access and skewed distribution are indicated in **Figure 2** and **Figure 3** respectively.

Each of the bars plotted for throughput have additional error bars to indicate the 99% confidence interval for the distribution across the multiple iterations of the experiment performed.

The different scenarios enumerated in the graphs are described below:

- **Local:** All requests for keys made are served by the key-value store on the local node. This is the theoretically ideal scenario.
- **Remote:** All requests for keys made are served not available on the local key-value store, and for each request, the penalty of having to retrieve the key's value from a remote node, is incurred.
- **Optimized:** All requests for keys made are served not available on the local key-value store. However, as the requests keep being made, the usage statistics are logged, and the Redynis daemon, following the Ownership coefficient policy detailed in **section 6.1**, replicates the keys to the local key-value store on the fly, to mitigate the penalty incurred by having to make remote requests for a frequently accessed key.

The hypothesis being tested by the experiment is to examine if the optimized option is a good-enough alternative to a naive global replication of all keys across all nodes in the key-value cluster. The experimental results corroborate this hypothesis.

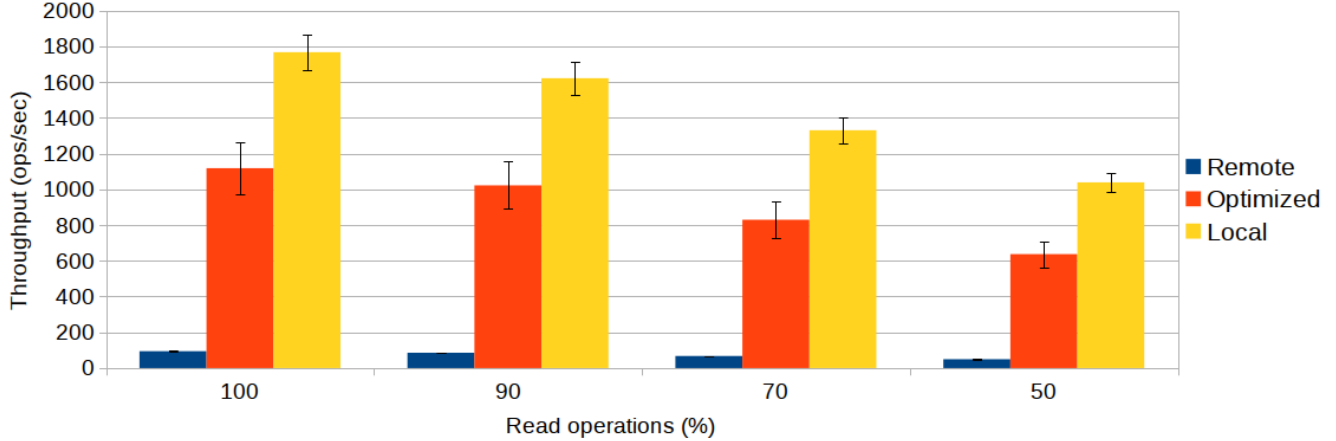


Figure 2: Uniform Object Access Distribution

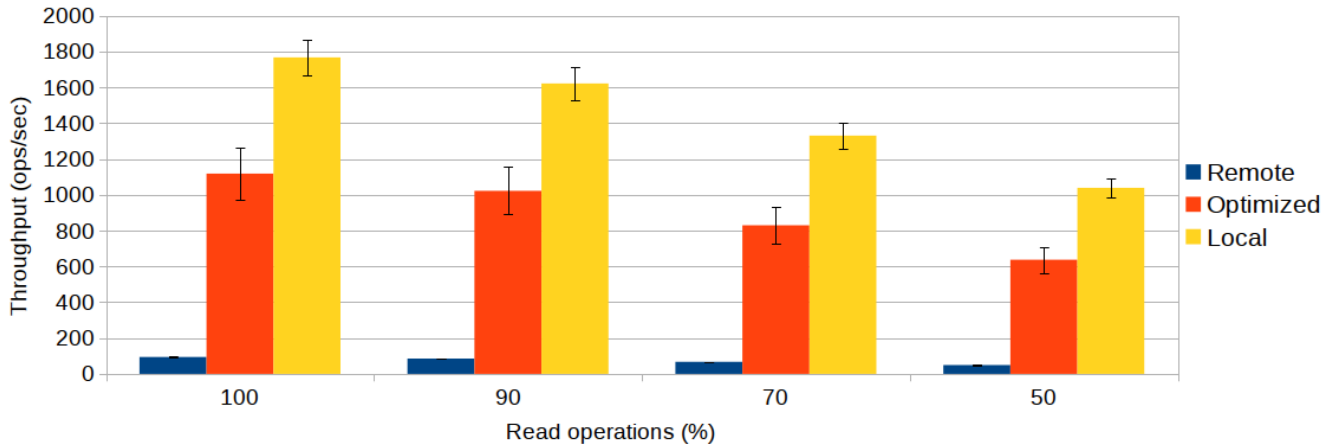


Figure 3: Skewed Object Access Distribution

10. CONCLUSIONS

The performance of the system designed for the experiment is approximately ten-fold better than the scenario in which all the requests made are re-routed to remote nodes. It is also nearly comparable to the theoretical ideal key-value store that contains all the keys, which could prove to be a helpful alternative to having a fully-replicated Redis cluster. The experimental results point to the hypothesis being proven true.

The Redynis system expands the feature set of an already widely used key-value store, and permits usage of a shared-nothing, independent set of Redis nodes as a shared-something cluster, to enable intelligent partitioning of data. This is particularly helpful in use-cases that require a widely geo-distributed set of key-value store nodes, and there are main memory constraints on the hardware specifications for the nodes the key-value stores are deployed on. The experimental performance is indicative of what the system has to offer. All of the above is offered while still maintaining strong serializability guarantees for the write operations on

the distributed cluster.

11. FUTURE WORK

This section lists the threads of future work that are envisioned for Redynis.

- The current architecture doesn't respond well to a failure of the master propagator, on which the write serialization depends upon. Future work would primarily be focused on implementing failure handling mechanisms. These mechanisms can be introduced into the existing framework using a heartbeat mechanism to detect when the master propagator goes offline, and electing another node to take its place.
- The data placement strategy that is currently in use is fairly trivial. A more sophisticated placement computation model can be plugged into RedynisDaemon, in its stead, for more accurate placement decision strategies.

- The RESTful web-service wrapper in the existing implementation is only responsible to aggregating metrics and directing client requests. It can be extended to form a framework for additional data-collection, which can, in turn be used to build predictive models that can identify patterns in data accesses, and pre-emptively move data based on the features of the model learnt.

12. ACKNOWLEDGMENTS

The author would like to thank Dr. Khuzaima Daudjee for his guidance, suggestions and feedback during the conceptualization of this research project.

13. REFERENCES

- [1] Ip latency statistics. <http://www.verizonenterprise.com/about/network/latency>.
- [2] Redis. <http://redis.io>.
- [3] Stackoverflow developer survey. <http://stackoverflow.com/research/developer-survey-2016>.
- [4] O. Asad and B. Kemme. Adaptcache: Adaptive data partitioning and migration for distributed object caches. In *Proceedings of the 17th International Middleware Conference*, page 7. ACM, 2016.
- [5] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.
- [6] V. John. Redynis daemon. <https://github.com/Redynis/RedynisDaemon>.
- [7] V. John. Redynis service. <https://github.com/Redynis/RedynisService>.
- [8] J. Pan, Y. T. Hou, and B. Li. An overview of dns-based server selections in content distribution networks. *Computer Networks*, 43(6):695–711, 2003.
- [9] A. Quamar, K. A. Kumar, and A. Deshpande. Sword: scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 430–441. ACM, 2013.
- [10] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulmaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.